ELECTRONIC PUBLISHING, VOL. 8(2 & 3), 153–165 (JUNE & SEPTEMBER 1995)

PHILIP N. SMITH AND DAVID F. BRAILSFORD

Electronic Publishing Research Group
Department of Computer Science
University of Nottingham
University Park
Nottingham NG7 2RD, UK

e-mail: dfb@cs.nott.ac.uk

SUMMARY

The Portable Document Format (PDF), defined by Adobe Systems Inc. as the basis of its Acrobat product range, is discussed in some detail. Particular emphasis is given to its flexible object-oriented structure, which has yet to be fully exploited. It is currently used to represent not logical structure but simply a series of pages and associated resources.

A definition of an Encapsulated PDF (EPDF) is presented, in which EPDF blocks carry with them their own resource requirements, together with geometrical and logical information. A block formatter called Juggler is described which can lay out EPDF blocks from various sources onto new pages.

Future revisions of PDF supporting uniquely-named EPDF blocks tagged with semantic information would assist in composite-page makeup and could even lead to fully revisable PDF.

KEY WORDS PDF Object-oriented Blocks Structured documents Juggler

1 INTRODUCTION

Over the last three years, a number of vendors have launched applications and file formats variously referred to as ‘digital paper’ or ‘portable documents’. The three most common examples are Novell’s *Envoy* [1], Common Ground’s *Common Ground* [2], and Adobe’s *Acrobat* [3,4]. All three aim to provide a platform- and application-independent way of distributing electronically-produced final form documents. In each case, the principal components are a pseudo printer driver and a viewer for each supported platform. The printer driver enables any application to produce a file which can subsequently be viewed using the appropriate viewer software.

All three systems attempt to recreate, in electronic form, the ‘look and feel’ of the ink-on-paper document. Principal differences between the systems lie in their solutions to ‘the font problem’ (i.e. what to do if the viewer software doesn’t have the fonts used in the document) and their provision of ‘added value’ features such as hypertext links, tables of contents, annotations, text highlighting, searching etc. This paper does not set out to analyse the relative technical merits of these rival systems, but it seems clear that Acrobat and its underlying Portable Document Format (PDF) are winning the race to become a *de facto* standard. That fashionable measure of success, presence on the Internet, certainly bears

CCC 0894–3982/95/020153–13

Received 3 April 1996
Revised 28 June 1996

witness to this, with hundreds of World Wide Web sites now having PDF files publicly available,[1] at least two public mailing lists for discussion, and an Internet newsgroup (`comp.text.pdf`). Companies such as AT&T and Intel Corporation make extensive use of PDF [5], and the free Acrobat Reader viewer can be found on many CD-ROMs as well as at various sites around the Internet. Perhaps most significantly, because the format has been published [6], clones of Acrobat are beginning to appear [7,8]. In L. Peter Deutsch's words, 'Implementation is the sincerest form of flattery.' [9]

PDF is particularly interesting because it is based on Adobe's well-regarded PostScript (level 2) page description language. For faster access in viewer software, PDF pages are arranged in a tree structure, instead of linearly as in most PostScript files. There are several standard 'hyper-features', including links and bookmarks, and new features can be added to Acrobat software by third-party plug-ins written using the Acrobat SDK. There are various compression options including JPEG and fax compression for images and LZW compression for almost anything.

PDF can be generated by a pseudo printer driver called *PDFWriter* or directly from PostScript using Adobe's *Distiller*, which is a full level 2 PostScript interpreter. Distiller is useful on platforms (such as UNIX) which do not have a standard printer driver interface, for documents containing Encapsulated PostScript (EPS) [10] figures or simply for documents which exist only in PostScript form. The added-value features of PDF can be created using a special PostScript operator called pdfmark — see Smith *et al*. [11] for a description of automated link generation using this operator. Though PDF files can be produced from PostScript, PDF is *not* just a tidied-up version of PostScript. The next section describes the internals of a PDF file in some detail. It is the purpose of the research described here to exploit the object-oriented structure of PDF, so that more abstract notions of block-structured documents can be combined with its display-oriented document model.

2 INSIDE PDF

Version 1.0 of PDF was published in 1993 [6] in what has come to be known as the 'putty book.'[2] A number of backwardly-compatible updates were made for version 1.1 and these are defined in the revised version of the PDF Reference Manual [12]. We give here a general description of the format and then describe, in more detail, those aspects relevant to later sections.

Whereas a PostScript file is essentially a continuous stream of commands to print an ordered sequence of pages, PDF files consist of a number of distinct *indirect objects* which may be distributed randomly within the file. Each object is given a number and is indexed by a cross-reference table at the end of the file. The cross-reference table gives the byte offset of the start of each object, so the whole file does not have to be read in order to locate all its objects. An *indirect reference* is a reference to an object by number — effectively a pointer to that object. Every PDF document has a root object (Catalog) containing indirect references to various structures, the most important of which is the pages tree. This structure is illustrated in figure 1. The pages tree is a balanced tree of Pages objects, with Page objects at the leaves, and allows faster access to documents containing hundreds — possibly thousands — of pages.

[1] A recent (6th March 1996) Webcrawler search returned 2635 hits for PDF or Acrobat

[2] This refers to the book's colour. It is mischievously rumoured that the cover was meant to be gold, but didn't quite make it.

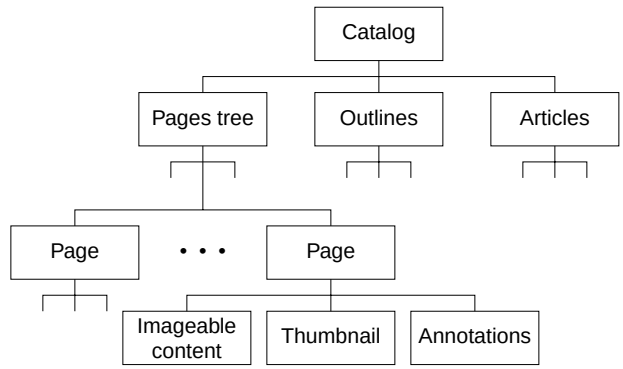

Figure 1. Part of the main PDF file structure

An indirect object may contain any single PostScript-type object such as a number, string, array or dictionary. A dictionary is just a set of key–value pairs where a value is accessed associatively by giving its key. Thus, a dictionary may contain any number of PostScript-type objects, and most indirect objects within a file are dictionaries storing chunks of related information. PDF also defines an object type *stream* which is a stream of data preceded by a dictionary declaring its length and any filters required to decode it.

```
10 0 obj
<<
/Type /Page
/MediaBox [0 0 612 792]
/Parent 4 0 R
/Resources <<
  /ProcSet [/PDF /Text]
  /Font << /F1 20 0 R /F2 22 0 R >>
>>
/Contents 11 0 R
/Thumb 10 0 R
/Annots [13 0 R 15 0 R]
>>
endobj
```

Figure 2. A typical page object

An example of a Page object (a leaf node in the Pages tree) is shown in figure 2. A Page object is a dictionary with a Parent key linking it into the pages tree. The `4 0 R` is a reference to object number 4 0.[3] Each Pages object (internal nodes of the pages tree) has an array of references to its Kids. The Resources dictionary lists fonts, large images and PostScript-type forms (frequently-called sequences of instructions) used by the page.

[3] The second number here is the *generation number* which has significance only after a PDF file has been incrementally updated. A single object number suffices for the remainder of this discussion.

```
11 0 obj
<< /Length 66 >>
stream
BT
/F1 10 Tf
1 0 0 1 100 700 Tm
(A short line of 10pt text.)Tj
ET
endstream
endobj
```

Figure 3. A simple contents stream object

A font resource can be an embedded Type 1, Type 3 or TrueType font, or a font descriptor enabling Multiple Master technology [13] to be used to generate a metric-compatible font at viewing time.

The page is imaged by the Contents stream. It is important to note that the Contents may also be an array of streams. This option is never taken by PDFWriter or Distiller when converting whole pages to PDF, but it is vital to our later definition of EPDF.

The Contents stream is the part of a PDF file that really looks like PostScript. However, PDF does not contain any of the programming constructs such as procedures, loops and conditionals that may be used in PostScript. In fact it is very similar to the stylised PostScript that is Adobe Illustrator's file format [14]. To convert PostScript to PDF, Distiller executes the PostScript program and writes out the results in terms of PDF's non-extensible set of page marking operators.

Although PDF is not a rich programming language, the *imaging model* supported by the page marking operators is just as powerful and device-independent as that of Level 2 PostScript. Consequently, vector graphics and text may be scaled to any magnification for viewing.

Figure 3 shows a short example of a Contents stream wherein the resources are refered to by the name assigned in the Resources dictionary. That is, `/F1 10 Tf` is used to select the font in object 20 (see figure 2) at a scale of 10 points.

Having resources such as fonts and images stored separately enables them to be associated with any of the pages in the document. Strangely, however, the putty book states that each resource must have a Name entry which must match the name in the Resources dictionary of any referring page. Thus, all pages using that resource must refer to it by the same name. This seems unnecessarily restrictive, and experimentation reveals that current Acrobat implementations do not enforce this rule.

The Thumb entry in a Page object is a tiny JPEG-compressed image of the page, which can be used for navigation in a viewer. More interesting is the Annots (annotations) dictionary which contains all the 'added value' features such as links and 'yellow stickies'. It should be noted that these features are associated, not with particular words or pictures on the page, but with areas of it. For instance, a link annotation contains a Rect key which is an array of four numbers defining the rectangle which the user must click to follow the link. Similarly, the destination is typically another bounding box, or a position and magnification.

Throughout a PDF file, any unrecognised dictionary key should simply be ignored by a viewing application. This enables third parties to add their own extensions which could, for instance, be handled by a plug-in written for the Acrobat Exchange Application Programmer's Interface (API). Many plug-ins are available [15,16] both commercially and for free.

3 BLOCK-BASED FORMATTING

This short section summarises an implementation of a block-based text formatter which gave rise to the idea of block-based PDF. Block-based document processing is now a well-known technique. Batch formatters such as *troff* [17] and TEX [18] (used to format this paper) build words into lines, and lines into pages. In a block-based system, the pagination process is left to a separate stage, after lines have been made into *blocks* such as paragraphs, tables and figures.

dlink [19,20] is a recent example which works with blocks of modified *ditroff* intermediate code (DIC). Modified versions of the *ms* macros [21] are used to format individual blocks which dlink then lays out onto pages. Knowing in advance the sizes of all the blocks enables dlink to produce better pagination and to avoid widows and orphans.

In order that dlink can lay out blocks correctly, they must contain certain pieces of information in addition to the DIC required to print them. This information includes the block's *type*, its *need*, its *height* and its *glue*. There are four types of block in the system:

1. A breakable block
2. An unbreakable block
3. A breakable float
4. An unbreakable float

If a breakable block (such as a paragraph) has to be broken because its *height* is larger than the remaining space on the page, suitable break points can be found by looking at the end-of-line markers in the DIC. The block's *need* specifies the minimum amount of the block which can be shown on its own at the bottom of a page (meaning that orphans can be avoided). Its *glue* specifies how much space should be left between the block and the preceding block.

The great virtue of DIC for dlink's processes is that line endings are clearly marked (which is not the case in TEX's DVI). Even so, ordinary DIC is not ideal because all coordinates are absolute, and this led to the development of a modified version of *ditroff* which generates DIC with relative motion commands. This means that one positioning command can be applied to reposition a block, with its contents and internal displacements intact.

Having got dlink working with this relative-motion DIC, a DVI to (modified) DIC converter was then written, enabling composite documents to be created from blocks obtained via LATEX and *troff*.

Unlike DIC, PDF can represent vector and bitmap graphics as well as plain (or not-so-plain) text. As explained in the previous section, it is also an object-based file structure, which can be extended by the insertion of additional dictionary keys. The rest of this paper considers how PDF might be extended to support a block-based document model, and presents one particular implementation.

4 BLOCK-BASED PDF

This section presents a method of using PDF to represent block-based documents. As we have seen, the finest level of granularity in PDF is usually the page. This means Acrobat Exchange can already support addition, removal and replacement of pages. Extending PDF to support a block model provides many benefits.

The term *Encapsulated PDF* (EPDF) is used to refer to a PDF file consisting of any number of *EPDF blocks*. EPDF blocks can be used like Encapsulated PostScript (EPS) figures if a document is destined for direct processing into PDF rather than PostScript. They can also be used as input to an EPDF-based block formatting system called Juggler. Any EPDF file is a valid PDF file and so can be viewed by existing PDF viewers.

4.1 Encapsulated PDF (EPDF) blocks

An Encapsulated PostScript (EPS) [10] file is a single page PostScript file which may contain almost arbitrary PostScript.[4] EPS files can be included in PostScript document files, and are often used for pictures and diagrams where an application does not directly support such features. A number of structured comments in an EPS file provide extra information to the application. The most important of these is the %%BoundingBox: comment which states, in the default coordinate system, where on the page the figure will, by default, be printed. This provides enough information for an application to place a figure correctly, by scaling and translating the coordinate system before including the EPS file.

An Encapsulated PDF (EPDF) block may be compared with an EPS file. An example is shown in figure 4. The block is a stream object, very similar to an ordinary page's contents stream (figure 3) but with some extra information. It has a bounding box so that it can be positioned properly, and a type (para). The type of an EPDF block is more akin to element types in SGML [22] than to the layout-oriented blocks and floats in dlink. Other types might be sectionhead and picture.

Recalling that the PDF specification requires a Page to declare all the resources it uses, it follows that a Page containing many EPDF blocks (see later description) must declare all the resources required by its constituent blocks. For this reason, EPDF blocks declare the resources they require in their own Resources dictionary. If a block is to be extracted and used elsewhere, only the resources in the block's Resources dictionary need to be carried over with it. There, when the block is inserted into a new page, its resources must be added to the new Page object's resources.

These extra entries in the block stream's dictionary are ignored by standard PDF viewers but are used by Juggler for block-based formatting.

4.2 Pages of EPDF blocks

We see, therefore, that an (E)PDF file may contain any number of these EPDFBlock objects and their associated resources. Each Page object (see figure 2) in the file references all the blocks it requires using the Contents entry. This is possible because the Contents may be an array of references to streams instead of just one reference to a single stream. For example, the entry

[4] There are a number of rules and recommendations for certain PostScript operators to avoid unwanted side effects when an EPS file is included in another PostScript file.

```
18 0 obj
<<
/Type /EPDFBlock
/Subtype /para
/BoundingBox [90 500 450 560]
/Resources <<
  /ProcSet [/PDF /Text]
  /Font << /F4 9 0 R >>
>>
/Length 19 0 R
>>
stream
...
endstream
endobj
```

Figure 4. An EPDF block of type 'para'

```
/Contents [ 40 0 R 41 0 R 42 0 R 43 0 R 44 0 R ]
```

would cause the viewer to execute the streams (EPDF blocks) 40 through 44 in order to build up the page, just as if they were merged into one stream. Importantly, the graphics state (the current transformation matrix, colour, line width etc.) is not changed between streams.

Blocks may be positioned by interleaving references to them with references to *positioner objects*. These are small streams which simply translate the coordinate system[5] appropriately, ready for execution of the next block's stream.

4.3 Repeated blocks

It is quite possible for an EPDFBlock to be referenced more than once. For example, a logo or watermark could be referenced by every page in the document. This method of sharing a common copy of an object, rather than replicating it on every page, can lead to significant file size reductions. If a block needs to be placed or scaled differently, it simply requires a different positioner object.

5 JUGGLER

Juggler is a batch formatter which extracts blocks from any number of EPDF files and lays them out onto new pages to create a new EPDF file. The layout process is controlled by a style file which defines frames on pages, and says which blocks to select from which files and in what order. The style file also says what the spacing should be between blocks of different types, and where or whether they may be split. This paper does not discuss the style file in great detail, but some parts are mentioned to clarify the operation of the Juggler program itself.

[5] Actually, they also do some saving and restoring of the graphics state, but this is a minor detail.

Juggler builds up pages one at a time by taking blocks from the input files in the order specified by the style file. To add a block to a page, the following steps are taken.

1. Copy across the block object, all its resource objects and all the objects to which they refer. In the process, change all the object numbers to fit into the new file.
2. Add the block's resources to the Page object's Resources dictionary, avoiding name clashes (see section 5.2).
3. Generate an appropriate positioner object.
4. Add references to the positioner object and the block to the Page object's Contents array.

If the block has already been placed into the new file (for example because it is a letterhead which appeared on the previous page) then there is no need for it to be copied again — step 1 can be omitted.

5.1 Visual splitting

A block at the bottom of a column will often have to be split and continued in the next column. dlink is able to split breakable blocks because it is easy to tell from the DIC where each line ends. However, the PDF in an EPDF block stream need not be that well organised. As in PostScript, there are infinite ways to describe the appearance of a page. To take an extreme example, all the 'a's could be shown, then all the 'b's, all the 'c's and so on until the block is complete. Even in the clean, well-structured PDF generated by Distiller and PDFWriter, there are no end of line markers.

Rather than attempting to parse the stream and re-write two smaller streams, Juggler generates two references to the complete stream. The positioner object preceding the first reference crops out the bottom of the block, and that preceding the second crops out the top. This may seem clumsy, and does require careful specification of bounding boxes, but also has one major advantage: the block remains intact. Even though it appears to be split, it is still a single object; a complete, self-contained logical entity. It could be extracted from the new file to be used in another, and from that to be used in another, and still remain essentially unchanged. It is interesting to compare this with the formatting process in ODA [23] which requires a content portion to be divided if it has to be formatted across columns.

In the current implementation of Juggler, split points are given in the style file on a per-object-type basis. For example, a typical entry might be

```
/Splits <<
  /para [ 24 12 36 ]
>>
```

indicating that objects of type para may be split 24 points from the top, 36 points from the bottom or at any 12 point interval in between.

'Top' and 'bottom' are, of course, determined by the block's BoundingBox entry which must therefore be very carefully and consistently defined. Otherwise, split points will fall *within* lines, causing letters to be chopped off half way down. Overlapping ascenders and descenders will also cause ugly results, so in such special cases, it is not recommended to define split points.

5.2 Resource renaming

Occasionally, Juggler has to rename one or more of a block's resources to meet PDF's requirement that all resources used by a page must be given a unique name. Consider two EPDF blocks, each from a different file. The first uses the name F1 to refer to the Times Roman font, but the second uses F1 for Helvetica. If these are to be put on the same page, one of them needs to use a different name. Juggler automatically generates a new name, and in the second block replaces all occurrences of the name F1 with this new name.

This solves the problem for the above case, but further clashes can still arise. If a logo is to appear on every page, Juggler normally puts one copy of the object in the file and references it from each page. For this to work, the block's resource names must be unique to the whole file. As Juggler only makes one pass it cannot ensure this, so if there is a name clash it writes another copy of the block with different resource names.

If resource names needed only to be unique to the streams in which they were used, none of this renaming would be necessary. However, as all resources must be declared in the Page object, all EPDF blocks on the page must share a common name space.

5.3 An example

Figure 5 shows a PDF file being viewed with Acrobat Reader. PostScript from LaTeX was distilled to produce PDF for the text blocks (the section heading, two subsection headings and five paragraphs). To force Distiller to make a separate stream for every block,

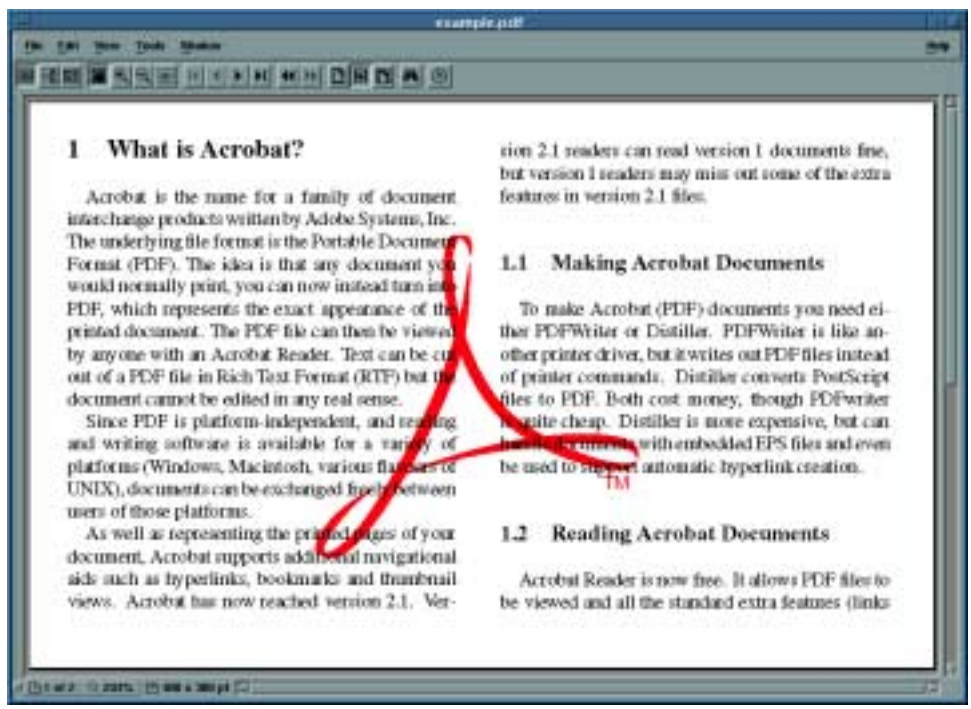

Figure 5. A PDF file produced by Juggler

```
54 0 obj
<<
/Type /Page
/Parent 3 0 R
/Contents [ 4 0 R 6 0 R 8 0 R 11 0 R 12 0 R 14 0 R 16 0 R
19 0 R 21 0 R 24 0 R 26 0 R 28 0 R 30 0 R 32 0 R 33 0 R
35 0 R 30 0 R 37 0 R 39 0 R 41 0 R 43 0 R 45 0 R 47 0 R
49 0 R 51 0 R 53 0 R ]
/Resources <<
/ProcSet [ /PDF /Text ]
/Font <<
/F1 9 0 R
/JugF1 17 0 R
/F2 22 0 R
>>
>>
>>
endobj
```

Figure 6. The page object produced by Juggler

each was placed on a separate page. The resulting PDF file was then edited to insert the extra EPDF information. The Acrobat logo came from a separate EPDF file.

Notice that the third (and fifth) paragraph has been split. In the PDF file the paragraph is in object 30, and so in the Contents entry of the Page object (see figure 6), there are two references to object 30.

Note also the font name JugF1. This was generated by Juggler because of a resource name clash. The logo file used F1 to refer to Helvetica (for the trademark sign) and the text file used F1 to refer to Times Bold for the headings. In the combined file, the streams for the headings are all re-written to refer to JugF1.

6 FURTHER POSSIBILITIES

The method of visual splitting means that blocks remain essentially unchanged as they are moved from one file to another. Giving blocks a unique ID would mean a block could be identified without reference to any particular file it might currently be stored in.

One of the ‘added value’ features of PDF is its support for hyperlinks between areas of pages. Having worked out the transformation necessary to position a block, Juggler could apply the same transformation to the sources of links so that they moved with the block. Destinations are more of a problem, but the existing *named destination* mechanism [12] could be used with unique block IDs.

There is a big difference between block-based documents and structured documents. In an EPDF file, there are many blocks, but there is no notion of the relationships between them. One possible extension would be to represent simple hierarchical relationships by giving blocks a Kids entry. Considering the example in figure 5, the section heading could have references to the first three paragraphs and the two subsection headings as its array

of kids. The two subsection headings would each have a paragraph as its child. Other mechanisms could be used, for instance to associate a paragraph block with a footnote block. Adding this would create a neat file structure consisting of a tree of blocks, a tree of pages and the page contents arrays providing the interface between the two. Also, telling Juggler to insert a block could cause it to insert all the kids as well.

7 PROBLEMS

Although the EPDF–Juggler mechanism works well, there are some implementation problems, and some more serious impediments to further progress.

The resource naming problem has already been discussed in section 5.2. The decision made in the design of PDF to declare resources at the page level and not the content stream level was probably made for performance reasons. It is interesting to note that PDF supports 'form' objects similar to those in Level 2 PostScript. In a PDF form (which is a resource of any page which uses it), the resources required *by the form* can optionally be declared in the form's dictionary, but they *must* also be declared at the page level, where their names will have to be unique.

An alternative definition of EPDF might extend form objects instead of content streams, and call all the blocks (forms) on a page from within a single contents stream. This would be neater in many ways, but it is difficult to generate forms and in current Acrobat implementations, they are very much second-class objects. In particular, they are ignored by all the text search and highlighting tools.

Forms must specify a strict bounding box for what they image, and this might appear to tie in nicely with EPDF's need to specify a bounding box. However, the bounding boxes required by Juggler are not strict. In fact, for correct alignment and splitting, they must be defined relative to the text baselines. For 10/12 text one might define the top to be 10 points above the top baseline and the bottom to be 2 points below the bottom baseline. This way the block can be split at 12 point intervals without chopping off ascenders or descenders, and baselines can be lined up correctly in multi-column formats. This tricky requirement is imposed by the fact that the actual contents of the stream are ignored.

In fact, this is the biggest stumbling block of all. Ignoring the contents makes it impossible to reformat for a different line length (measure) or to adjust section and cross-reference numbering. dlink was able to tackle the numbering problem by parsing the DIC for special markers. PDF would have to be written directly (i.e. not by distilling PostScript) to insert such markers into EPDF blocks as comments.

8 CONCLUSION

A PDF file contains a number of fixed pages through which a user can move in order to read the document. Our proposal for EPDF extends the underlying architecture so that the document's content consists of a pool of logical, typed blocks. In a particular EPDF file, these blocks will be paginated in a particular way, but Juggler allows them to be extracted and re-used in other documents. This mechanism has a number of different uses, for example:

- Documents can be reformatted for a different page length, for example to make them more like the 4×3 aspect ratio of a computer screen.

- Logos, watermarks and other recurring objects can be added programmatically and by reference, so reducing file size.
- Objects can be replaced without regenerating all the others. For instance, a black and white photograph could be replaced with a colour version.
- Blocks remain unchanged by the formatting process and so can be extracted from any available source document for reuse.
- The fact that blocks are typed means that more specific searches could be made, particularly if blocks are arranged hierarchically as suggested in section 6.

All these possibilities stem from the logical, block-based approach. However, the block model has its limits. Fundamentally, the content streams contain no information about the content they render. There exists the same problem with revising PDF content streams as with PostScript language programs: nothing is known of *why* the content is formatted the way it is, so it is not possible to alter and reformat it.

It is interesting to consider here Adobe Illustrator [14] which uses a highly stylised form of PostScript to represent, in a rather restricted way, the content's meaning as well as its appearance. By writing more specialised content streams, more functionality could be supported by EPDF. For instance, line markers could be inserted, and section numbers could be marked so that they could be revised. Juggler could then be told how to handle these extra features for a particular class of documents.

Such additions would go some way towards combining a document's logical structure with the expressive power of PDF's imaging model. We believe that future extensions to PDF should define EPDF so that blocks can be freely transported between files. Blocks should carry with them knowledge not just of their geometry but also of their meaning within a document, so that future generations of document production software can reuse them intelligently.

ACKNOWLEDGEMENTS

In 1995, one of us (PNS) made a three-month visit to the Acrobat Engineering group at Adobe Systems Inc. Particular thanks are due to Liz McQuarrie and Steve Zilles with whom many useful discussions were held over that period. Thanks also to Bob Wulff for his help in bringing the trip about.

REFERENCES

1. Tumbleweed Software Corporation. Envoy Viewer.
   URL: `http://www.twcorp.com/viewer.htm`.
2. Common Ground Software Inc. Common Ground Software.
   URL: `http://www.commonground.com/index.html`.
3. Adobe Systems Inc. Adobe Acrobat.
   URL: `http://www.adobe.com/acrobat/`.
4. Patrick Ames, *Beyond Paper — the official guide to Adobe Acrobat*, Prentice Hall, 1993. ISBN 1-56830-050-6.
5. Adobe Systems Inc. Adobe Customer Spotlights.
   URL: `http://www.adobe.com/studio/spotlights/#acrobat`.
6. Adobe Systems Inc., *Portable Document Format Reference Manual*, Addison-Wesley, Reading, Massachusetts, June 1993.
7. Zeon Corporation. DocuCom Series Product.
   URL: `http://www.zeon.com.tw/d-com.htm`.

8. GhostScript, GhostView & GSView.
URL: `http://www.cs.wisc.edu/~ghost/`.
9. L. Peter Deutsch. Private communication.
10. Adobe Systems Inc., *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, second edition, December 1990.
11. Philip N. Smith, David F. Brailsford, David R. Evans, Leon Harrison, Steve G. Probets, and Peter E. Sutton, 'Journal publishing with Acrobat: the CAJUN project', *Electronic Publishing—Origination, Dissemination and Design*, **6**(4), 481–493, (December 1993). Proceedings of the Fifth International Conference on Electronic Publishing, Document Manipulation and Typography (EP94).
12. Adobe Systems Inc., *Portable Document Format Reference Manual Version 1.1*, Adobe Systems Inc., Mountain View, California, March 1996.
URL: `http://www.adobe.com/supportservice/devrelations/PDFS/TN/PDFSPEC.PDF`.
13. Adobe Systems Inc., 'Type 1 Font Format Supplement', Technical Note 5015, Adobe Systems Inc., Mountain View, California, (15th January 1994).
URL: `http://www.adobe.com/supportservice/devrelations/PDFS/TN/5015.Type1_Supp.pdf`.
14. Adobe Systems Inc., *Adobe Illustrator Document Format*, Mountain View, California, July 1990.
15. Adobe Systems Inc. Free Adobe Acrobat Plug-Ins.
URL: `http://www.adobe.com/acrobat/plugins.html`.
16. Emerge. Acrobat Software Add-ons: Expand the Functionality.
URL: `http://www.emrg.com/down_sfwe.html`.
17. Joseph F. Ossanna, 'NROFF / TROFF user's manual', *Computing Science Technical Report No. 54*, (October 1976).
18. D. E. Knuth, *The TEXbook*, Addison-Wesley, Reading, Massachusetts, 1984.
19. M. J. Groves and D. F. Brailsford, 'Separate compilation of structured documents', *Electronic Publishing—Origination, Dissemination and Design*, **6**(4), 315–326, (December 1993). Proceedings of the Fifth International Conference on Electronic Publishing, Document Manipulation and Typography (EP94).
20. M. J. Groves, *Separate Compilation of Structured Documents*, Ph.D. dissertation, University of Nottingham, 1995.
21. M. E. Lesk, *Typing Documents on the* UNIX *System: Using the -ms Macros with Troff and Nroff*, Bell Laboratories, Murray Hill, New Jersey 07974, 1978.
22. Charles F. Goldfarb, *The SGML Handbook*, Oxford University Press, 1990.
23. ISO/DIS 8613 Information processing, *Office Document Architecture (ODA)*, 1986.